Error-resistant Implementation of DNA Computations *

Martyn Amos † Alan Gibbons
Department of Computer Science, University of Warwick, Coventry CV4 7AL, England

David Hodgson
Department of Biological Sciences, University of Warwick, Coventry CV4 7AL, England

Abstract

This paper introduces a new model of computation that employs the tools of molecular biology whose *in vitro* implementation is far more error-resistant than extant proposals. We describe an abstraction of the model which lends itself to natural algorithmic description, particularly for problems in the complexity class NP. In addition we describe a number of linear-time algorithms within our model, particularly for NP-complete problems. We describe an *in vitro* realisation of the model and conclude with a discussion of future work.

1 Introduction

The idea that living cells and molecular complexes can be viewed as potential machinic components dates back to the late 1950s, when Richard Feynman delivered his famous paper [4] describing "sub-microscopic" computers. More recently, several papers [1, 10, 16] (also see [7, 13]) have advocated the realisation of massively parallel computation using the techniques and chemistry of molecular biology. Adleman describes how a computationally intractable problem, known as the *directed Hamiltonian Path Problem* (HPP) might be solved using molecular methods. Recall that the HPP involves finding a path through a graph that visits each vertex exactly once. Adleman's method employs a simple, massively parallel random search. The algorithm is not executed on a traditional, silicon-based computer, but instead employs the "test-tube" technology of genetic engineering. By representing information as sequences of bases in DNA molecules, Adleman shows how existing DNA-manipulation techniques may be used to quickly detect and amplify desirable solutions to a given problem.

A recent attempt [9] to repeat Adleman's experiment has cast doubt upon the efficacy of extant models of DNA computation. The researchers performed Adleman's experiment twice; once on the original graph as a positive control, and again on a graph containing no Hamiltonian path as a negative control. The results obtained were inconclusive. The researchers state that *"At this time we have carried out every step of Adleman's experiment, but have not gotten an unambiguous final result."*

*Partially supported by the University of Warwick Research and Teaching Innovations Sub-Committee.

†martyn@dcs.warwick.ac.uk

The main purpose of this paper is to describe an alternative, but importantly, *feasible* model of DNA computation. The problem with extant proposals is that they assume that certain proposed biological operations are error-free. An operation central to most models is *extraction* of DNA strands containing a certain sequence. There are two problems with extraction. The first, and most important problem is that removal by DNA hybridization of strands containing the sequence is not 100% efficient[1], and may at times inadvertently remove strands that *do not* contain the specified sequence. Adleman did not encounter problems with extraction because only a few operations were required. However, for a large problem instance, the number of extractions required may run into hundreds, or even thousands. For example, a particular DNA-based algorithm may rely upon repeated "sifting" of a "soup" containing many strands, some encoding legal solutions to the given problem, but most encoding illegal ones. At each stage, we may wish to extract only strands that satisfy certain criteria (i.e., they contain a certain sequence). Only strands that satisfy the criteria at one stage go through to the next. At the end of the sifting process, we are hopefully left only with strands that encode legal solutions, since they satisfy all criteria. However, assuming 95% efficiency of the extraction process, after 100 extractions the probability of us being left with a soup containing (a) a strand encoding a legal solution, and (b) no strands encoding illegal solutions is about 0.006. Repetitive extraction will not guarantee 100% efficiency, since it is impossible to achieve the conditions whereby only correct hybridization occurs. Furthermore, as the length of the DNA strands being used increases, so does the probability of incorrect hybridization.

Clearly, for any non-trivial problem, reliance on the extraction operation must be minimised, or, ideally, removed entirely. In this paper we describe a novel model of DNA computation that obviates the need for hybridization extraction within the main body of the computation.

The rest of the paper is organised as follows. In Section 3 we describe a DNA implementation for which none of the problems described above exist, and which is therefore truly scalable. In Section 2 we describe an abstraction of the model which lends itself to natural algorithmic description. In addition we describe a number of algorithms within our model and discuss its computational power.

2 The Model of Computation

Here we describe an abstract model of computation which, as we show in Section 3, has a clean implementation in DNA chemistry. We note the initial contribution of Lipton [10] to the construction of such models.

Our model is particularly effective for algorithmic description. Moreover, it is sufficiently strong to solve any of the problems in the class NC which includes, of course, the notoriously intractable NP-complete problems. As we shall see, these problems naturally have polynomial-time (often linear-time) parallel solutions within the model. This usually comes with expense of exponentially large data sets.

Within the model, a computation consists of a sequence of operations on finite sets of strings. It is normally the case that a computation begins and terminates with a single set. Within the computation, by applying legal operations of a computation,

[1] The actual efficiency depends on the concentration of the reactants.

several sets may exist at the same time. We define legal operations on sets shortly but first consider the nature of an *initial set*.

An initial set consists of strings which are typically of length $O(n)$ where n is the problem size. As a subset, the initial set should include all possible solutions (each encoded by a string) to the problem to be solved. The point here is that the superset is supposed, in any implementation of the model, to be relatively easy to generate as a starting point for a computation. The computation then proceeds by filtering out strings which cannot be a solution. For example, if the problem is to generate a permutation of the integers $1 \ldots n$ then the initial set might include all strings of the form $p_1 i_1 p_2 i_2 \ldots p_n i_n$ where each i_k may be any of the integers in the range $[1 \ldots n]$ and p_k encodes the information "position k". Here, as will be typical for many computations, the set has cardinality which is exponential in the problem size. For our example of finding a permutation, we should filter out all strings in which the same integer appears in at least two locations p_k. Any of the remaining strings is then a legal solution to the problem. We return to this problem (whose solution, incidentally, may be regarded as a very useful standard operation within our model for the solution of other problems) after defining some basic legal operations on strings.

2.1 Basic set operations

Here we define the basic legal operations on sets within the model. Our choice is determined by what we know can be effectively implemented by very precise and complete chemical reactions within the DNA implementation. The operation set defined here provides the power we claim for the model but, of course, it might be augmented by additional operations in the future to allow greater conciseness of computation.

- *remove*$(U, \{S_i\})$. This operation removes from the set U, in parallel, any string which contains at least one occurrence of any of the substrings S_i.
- *union*$(\{U_i\}, U)$. This operation, in parallel, creates the set U which is the set union of the sets U_i.
- *copy*$(U, \{U_i\})$. In parallel, this operation produces a number of copies, U_i, of the set U.
- *select*(U). This operation selects an element of U uniformly at random, if U is the empty set then *empty* is returned.

From the point of view of establishing the parallel time-complexities of algorithms within the model, these basic set operations will be assumed to take *constant-time*. This is certainly what the DNA implementation described in Section 3 provides.

2.2 A First Algorithm

We now provide our first algorithmic description within the model. The problem solved is that of generating the set of all permutations of the integers 1 to n. The initial set and the filtering out of strings which are not permutations were essentially described earlier. Although not NP-complete, the problem does of course have exponential-sized input and output.

The algorithmic description below introduces a format that we utilise elsewhere. The particular device of copying a set (as in copy$(U, \{U_1, U_2, \ldots, U_n\})$) followed by parallel *remove* operations (as in the employment of remove($U_i, \{p_j\neg i, p_k i\}$)) is a very useful compound operation as we shall see in several later algorithmic descriptions. Indeed, it is precisely this use of *Parallel Filtering* that is at the core of most algorithms within the model. The only non-selfevident notation employed below is $\neg i$ to mean (in this context) any integer in the range i=1, 2, ... , n which is *not* equal to i.

- **Problem: Permutations**

  Generate the set P_n of all permutations of the integers $\{1, 2, \ldots, n\}$.

- **Solution**

  - *Input*: The input set U consists of all strings of the form $p_1 i_1 p_2 i_2 \ldots p_n i_n$ where, for all j, p_j uniquely encodes "position j" and each i_j is in $\{1, 2, \ldots, n\}$. Thus each string consists of n integers with (possibly) many occurences of the same integer.
  - *Algorithm*

    for $j = 1$ to n **do**
    begin
    copy($U, \{U_1, U_2, \ldots, U_n\}$)
    for i=1, 2, ... , n and all $k > j$
    in parallel do remove($U_i, \{p_j\neg i, p_k i\}$)
    union($\{U_1, U_2, \ldots, U_n\}, U$)
    end
    $P_n \leftarrow U$

  - *Complexity*: $O(n)$ parallel-time.

After the jth iteration of the **for** loop, the computation ensures that in the surviving strings the integer i_j is not duplicated at positions $k > j$ in the string. The integer i_j may be any in the set $\{1, 2, \ldots, n\}$ (which one it is depends in which of the sets U_i the containing string survived). At the end of the computation each of the surviving strings contains exactly one occurence of each integer in the set $\{1, 2, \ldots, n\}$ and so represents one of the possible permutations. Given the specified input, it is easy to see that P_n will be the set of all permutations of the first n natural numbers. As we shall see, production of the set P_n can be a useful subprocedure for other computations.

2.3 Algorithms for a selection of NP-complete problems.

We now describe a number of algorithms for graph-theoretic NP-complete problems (see [6], for example). Problems in the complexity class NP seem to have a natural expression and ease of solution within the model. We describe linear-time solutions although, of course, there is frequently an implication of an exponential number of processors available to execute any of the basic operations in unit time.

2.3.1 3-vertex-colourability

Our first problem concerns proper vertex colouring of a graph. In a proper colouring, colours are assigned to the vertices in such a way that no two adjacent vertices are similarly coloured. The problem of whether 3 colours are sufficient to achieve such a colouring for an arbitrary graph is NP-complete [6].

- **Problem: Three colouring**

  Given a graph $G = (V, E)$, find a 3-vertex-colouring if one exists, otherwise return the value *empty*.

- **Solution**

  - *Input*: The input set U consists of all strings of the form $p_1c_1p_2c_2 \ldots p_nc_n$ where $n = |V|$ is the number of vertices in the graph. Here, for all i, p_i uniquely encodes "position i" and each c_i is any one of the "colours" 1, 2 or 3. Each such string represents one possible assignment of colours to the vertices of the graph in which, for each i, colour c_i is assigned to vertex i.
  - *Algorithm*

    for $j = 1$ to n **do**
        begin
        copy$(U, \{U_1, U_2, U_3\})$
        for i=1, 2 and 3, and all k such that $(j, k) \in E$
                in parallel do remove$(U_i, \{p_j\neg i, p_k i\})$
        union$(\{U_1, U_2, U_3\}, U)$
        end
    select(U)

  - *Complexity*: $O(n)$ parallel time.

After the jth iteration of the **for** loop, the computation ensures that in the remaining strings vertex j (although it may be coloured 1, 2 or 3 depending on which of the sets U_i it survived in) has no adjacent vertices that are similarly coloured. Thus, when the algorithm terminates, U only encodes legal colourings if any exist. Indeed, every legal colouring will be represented in U.

2.3.2 Hamiltonian path

A Hamiltonian path between any two vertices u, v of a graph is a path that passes through every vertex in $V - \{u, v\}$ precisely once [6].

- **Problem: Hamiltonian path**

  Given a graph $G = (V, E)$ with n vertices, determine whether G contains a Hamiltonian path.

- **Solution**

  - *Input*: The input set U is the set P_n of all permutations of the integers from 1 to n as output from **Problem: Permutations**. An integer i at position p_k in such a permutation is interpreted as follows: the string represents a candidate solution to the problem in which vertex i is visited at step k.
  - *Algorithm*

    for $2 \le i \le n - 1$ and j, k such that $(j, k) \notin E$
        in parallel do remove $(U, \{jp_ik\})$
    select(U)

- *Complexity*: Constant parallel time given P_n.

In surviving strings there is an edge of the graph for each consecutive pair of vertices in the string. Since the string is also a permutation of the vertex set it must also be a Hamiltonian path. Of course, U will contain every legal solution to the problem.

2.3.3 Subgraph isomorphism

Given two graphs G_1 and G_2 the following algorithm determines whether G_2 is a subgraph of G_1.

- **Problem: Subgraph isomorphism**

  Is $G_2 = (V_2, E_2)$ a subgraph of $G_1 = (V_1, E_1)$? By $\{v_1, v_2, \ldots, v_s\}$ we denote the vertex set of G_1, similarly the vertex set of G_2 is $\{u_1, u_2, \ldots, u_t\}$ where, without loss of generality, we take $t \leq s$.

- **Solution**

  - *Input*: The input set U is the set P_s of permutations output from the Permutations algorithm. For $1 \leq j \leq t$ an element $p_1 i_1 p_2 i_2 \ldots p_s i_s$ of P_s is interpreted as associating vertex $p_j \in \{u_1, u_2, \ldots, u_t\}$ with vertex $i_j \in \{v_1, v_2, \ldots, v_s\}$. The algorithm is designed to remove any element which maps vertices in V_1 to vertices in V_2 in a way which does not reflect the requirement that if $(p_s, p_t) \in E_1$ then $(i_s, i_t) \in E_2$.

  - *Algorithm*

    for j=1 to t **do**
        begin
        copy($U, \{U-1, U_2, \ldots, U_t\}$)
        for all $l, j < l \leq t$ such that $(p_j, p_l) \in E_2$ and $(ij, i_l) \notin E_1$
            in parallel do remove($U_j, \{p_l i_l\}$)
        union($\{U-1, U_2, \ldots, U_t\}, U$)
        end
    select(U)

  - *Complexity*: O($|V_s|$) parallel time.

For any remaining strings, the first t pairs $p_l i_l$ represent a one-to-one association of the vertices of G_1 with the vertices of G_2 indicating the subgraph of G_1 which is isomorphic to G_2. If select(U) returns the value *empty* then G_2 is not a subgraph of G_1.

2.3.4 Maximum clique and maximum independent set

A clique K_i is the complete graph on i vertices [6]. The problem of finding a maximum independent set is closely related to the maximum clique problem.

- **Problem: Maximum clique**

  Given a graph $G = (V, E)$ determine the largest i such that K_i is a subgraph of G. Here K_i is the complete graph on i vertices.

- **Solution**
  - In parallel run the subgraph isomorphism algorithm for pairs of graphs (G, K_i) for $2 \leq i \leq n$. The largest value of i for which a non- *empty* result is obtained solves the problem.
  - *Complexity*: $O(|V|)$ parallel time.

    A maximum independent set is a subset of vertices of a graph such that no two members of the set are adjacent [6].
- **Problem: Maximum independent set**

  Given a graph $G = (V, E)$ determine the largest i such that there is a set of i vertices in which no pair are adjacent.
- **Solution**

  Run the maximum clique algorithm on the *complement* of G.
  - *Complexity*: $O(|V|)$ parallel time.

2.4 Computational power of the model

It is clear that a lower bound on the computational power of the model is that it can solve any problem in the complexity class NP. This follows from the fact that we were able to encode solutions for various NP-complete problems for it and, of course, any instance of any problem in NP may be described as an instance of any NP-complete problem. Our examples also demonstrate that individual NP-complete problems seem to have a natural and direct encoding for the model.

It might be that the Parallel Random Access Machine (P-RAM [5]) or the Turing Machine can be directly simulated within the model which would establish that any algorithm can be realised by it. We have yet to pursue this line of research.

3 Implementation of the model in DNA chemistry

Notice that the algorithms of the previous section work perfectly well if the basic data structure, *set*, is replaced by *multiset*. The **permutation** algorithm now outputs a multiset in which each permutation appears as many times as it was represented in the input set. However, since the *select* operation returns a single element, the output of the other algorithms is exactly as before.

In the proposed implementation outlined below, the algorithms are realised by multisets of single-stranded DNA. In practice, to a very good approximation, there would be the same number of copies of each element in any such multiset.

3.1 The structure and enzyme manipulation of DNA

DNA (deoxyribonucleic acid) [3] encodes the genetic information of cellular organisms. It consists of *polymer chains*, commonly referred to as DNA *strands*. Each strand may be viewed as a chain of *nucleotides*, or *bases*. An n-letter sequence of consecutive bases is known as an n-mer or an *oligonucleotide* of length n. The four DNA nucleotides are adenine, guanine, cytosine and thymine, commonly abbreviated to A, G, C and T respectively. Each strand has, according to chemical convention, a $5'$ and a $3'$ end, thus

```
5’ A T A G A G T T 3’
   | | | | | | | |
3’ T A T C T C A A 5’
```

Figure 1: Structure of double-stranded DNA

```
5’ A T A G A G T T 3’
           | | |        (a)
3’         T C A    5’

5’ A T A G A G T T 3’
   | | | | | | |        (b)
3’ T A T C T C A    5’
```

Figure 2: (a) Primer anneals to longer template (b) Polymerase extends primer in the $5'$ to $3'$ direction

any single strand has a natural orientation. The classical double helix of DNA is formed when two separate strands bond. Bonding occurs by the pairwise attraction of bases; A bonds with T and G bonds with C. The pairs (A,T) and (G,C) are therefore known as *complementary* base pairs. In what follows we adopt the following convention: if x denotes an oligonucleotide, then $\overline{x}$ denotes the complement of x. The bonding process, known as *annealing*, is fundamental to our implementation. A strand will only anneal to its complement if they have opposite polarities. Therefore, one strand of the double helix extends from $5'$ to $3'$, and the other from $3'$ to $5'$, as, for example, in Figure 1.

In order to manipulate DNA strands and thus implement our operations we use specific classes of *enzymes*, each of which performs one or more specific tasks:

1. The DNA *polymerases* perform several functions, including the repair and duplication of DNA. Given a short *primer* oligonucleotide, p in the presence of nucleoside triphosphates, the polymerase extends p if and only if p is bound to a longer *template* oligonucleotide, t. For example, in Figure 2(a)), p is the oligonucleotide TCA which is bound to t, $ATAGAGTT$. In the presence of the polymerase, p is extended by a complementary strand of bases to the $3'$ end of t (Figure 2(b)).
2. *Restriction enzymes* [17, page 33] recognize a specific sequence of DNA, known as a *restriction site*. Any *double-stranded* DNA that contains the restriction site within its sequence is cut by the enzyme at that point[2] For example, the double-stranded DNA in Figure 3(a) is cut by restriction enzyme *Sau*3AI, which recognizes the restriction site $GATC$. The resulting DNA is depicted in Figure 3(b). In our implementation we use *Sau*3AI.

[2]In reality, only certain enzymes cut specifically at the restriction site, but we take this factor into account when selecting an enzyme.

```
            Sau3AI
               ↓
5' G G A T G A T C G G T A 3'
   | | | | | | | | | | | |              (a)
3' C C T A C T A G C C A T 5'

5' G G A T           G A T C G G T A 3'
   | | | |                   | | | |    (b)
3' C C T A C T A G           C C A T 5'
```

Figure 3: (a) Double-stranded DNA being cut by *Sau*3AI (b) The result

3. In double-stranded DNA, if one of the single strands contains a discontinuity (i.e., one nucleotide is not bonded to its neighbour) then this may be repaired by DNA *ligase* [2]. This allows us to create a unified strand from several bound together by their respective complements.

Another useful method of manipulating DNA is the Polymerase Chain Reaction, or PCR [11, 12]. PCR is a process that quickly amplifies the amount of DNA in a given solution. Each cycle of the reaction doubles the quantity of each strand, giving an exponential growth in the number of strands.

3.2 Implementation of the model

Here we first describe how an initial multiset within the model may be constructed in DNA, and then how the set operations may be implemented.

An essential difficulty in the model is that initial multisets generally have a cardinality which is exponential in the problem size. It would be too costly in time, therefore, to generate these individually. What we do in practice is to construct an initial solution, or *tube*, containing a polynomial number of distinct strands. The design of these strands ensures that the exponentially large initial multisets of our model will be generated automatically. The following paragraph describes this process in detail.

Consider an initial set of all elements of the form $p_1k1, p_2k_2, \ldots, p_nk_n$. This may be constructed as follows. We generate an oligo uniquely encoding each possible subsequence p_ik_i where $1 \leq i \leq n$ and $1 \leq k_i \leq k$. Embedded within the sequence representing p_i is our chosen restriction site. There are thus nk distinct oligos of this form. The task now is how to combine these to form the desired initial multiset. This is achieved as follows. For each pair $(p_ik_i, p_{i+1}k_{i+1})$ we construct an oligo which is the concatenation of the complement of the second half of the oligo representing p_ik_i and the complement of the first half of the oligo representing $p_{i+1}k_{i+1}$. We also construct oligos that are the complement of the first half of the oligo representing p_1k_1 and the last half of the oligo representing p_nk_n. There is therefore a total of $nk+(nk/2)+2$ oligos in solution. The *in vitro* effect of adding adding these new oligos is that the tube will now contain double-strands of DNA, one strand in each will be an element of the desired initial set. The new oligos have, through annealing, acted as "splints" to join the first oligos in the desired sequences. What we really require in solution are only the single strands encoding elements of the initial set. This is achieved as follows. We ligate the double strands and then heat the tube to break the hydrogen bonds between the encoding strands and the splint strands. The splint strands are created with magnetic beads attached to them in order to facilitate their removal at this stage.

It remains in our implementation to describe how the set operations are realized. This is achieved as follows:

- **Remove**

  remove is implemented as a composite operation, comprised of the following:

  - $mark(U, S)$. This operation marks all strings in the set U which contains at least one occurrence of the substring S.
  - $destroy(U)$. This operation removes all marked strings from U.

  mark is implemented by adding to U many copies of a primer corresponding to $\overline{S}$. This primer only anneals to single strands containing the subsequence S. We then add DNA polymerase to extend the primers once they have annealed, making double-stranded only the single strands containing S.

  We may then *destroy* strands containing S by adding the restriction enzyme Sau3AI. Double-stranded DNA (i.e. strands marked as containing S) is cut at the restriction sites embedded within, single strands remaining intact. We may then remove all intact strands by separating on length using *gel electrophoresis* [2]. However, this is not strictly necessary, and leaving the fragmented strands in solution will not affect the operation of the algorithm.

- **Union**

  We may obtain the *union* of two or more tubes by simply mixing them together, forming a single tube.

- **Copy**

  We obtain i "copies" of the set U by splitting U into i tubes of equal volume. We assume that, since the initial tube contains multiple copies of each candidate strand, each tube will also contain many copies.

- **Select**

  We can easily detect remaining DNA using PCR and then sequence strands to reveal the encoded solution to the given problem. One problem with this method is that there are often multiple correct solutions left in the soup which must be sequenced using nested PCR. A possible solution is to utilise *cloning*. The strands are designed such that they can be ligated into a double stranded DNA vector which, upon DNA transfection of a suitable bacterial host, generates single stranded DNA bacteriophage clones, each of which encodes a single solution. Each clone is represented as a single bacteriophage plaque on a lawn of the bacterial host. Tens of thousands of plaques can be present on a single lawn of bacteria and a hundred such lawns can be prepared from a single ligation.

  The plaques are pooled and single stranded DNA isolated. These single strands provide the substrate for oligonucleotide primer annealing, subsequent DNA polymerase extension of double strands and restriction enzyme digestion of any such double stranded DNA. This removes any illegal solutions, as described previously. Legal solutions are collected by retransfecting host bacteria, and again individual plaques represent one possible solution. Only intact bacteriophage DNA molecules are capable of transfecting the host bacterium. The "legal" plaques are again pooled and single stranded DNA isolated and subjected to further rounds of DNA extension and restriction and host transfection. The final collection of plaques are individually picked, then the DNA is isolated and subjected to standard DNA sequencing reactions.

Although the initial tube contain multiple copies of each strand, after many *remove* operations the volume of material may be depleted below an acceptable empirical level. This difficulty can be avoided by periodic amplification by PCR.

4 Advantages of the model

As we have shown, algorithms within our model perform successive "filtering" operations, keeping *good* strands (i.e., strands encoding a legal solution to the given problem) and destroying *bad* strands (i.e., those that do not). So long as the operations work correctly, the final set of strands will only consist of good solutions. However, as we have already stated, errors can take place. If either good strands are accidentally destroyed or bad strands are left to survive through to the final set then the algorithm will fail. The main advantage of our model is that it doesn't repeatedly use the notoriously error-prone separation by DNA hybridization method to extract strands containing a certain subsequence. Restriction enzymes are *guaranteed* to cut any double- stranded DNA containing the appropriate restriction site, whereas hybridization separation is never 100% efficient. Instead of extracting *most* strands containing a certain subsequence we simply destroy *all* of them with absolute certainty, without harming those strands that *do not* contain the subsequence.

5 Fundamental limitations on DNA computations

Although our implementation is much less error-prone than those previously proposed, there still exists at least one barrier to scalable DNA computation. This concerns the sheer weight of DNA required to solve any problems of large size in the model described in this paper. As Hartmanis points out in [8], if Adleman's experiment were scaled up to 200 vertices the weight of DNA required would exceed that of the Earth. DNA computations, like all others, cannot escape the "exponential curse". The approach of generating all possible solutions to a given problem and then gradually filtering out illegal solutions seems impractical for problem sizes beyond a moderate threshold. It is interesting to note, however, that estimates lead us to believe that our model will perform more rapid computations than conventional computers on a range of feasible problem sizes. One possible way around the weight problem is to look for implementation of existing computing paradigms in DNA. Reif [14] points to one possible way forward, describing a method for simulating a CREW P-RAM *in vitro*. However, Reif uses the error-prone hybridization separation method described earlier, which we believe may be avoided by our methodology. Another development is the proposed [15] DNA and restriction enzyme implementation of Turing Machines.

The Turing Machine is, however, sequential in nature, and so the implementation of this does not exploit the inherent parallelism of DNA computation. It does however establish formally that any algorithm can be realised through DNA.

6 Conclusions

Recent work in DNA computation has cast serious doubt on the reliability of extant models. The use of error-prone operations such as DNA hybridization separation can lead to inconclusive final results. In this paper we described a feasible model of DNA computation that avoids the problems of those previously proposed. An abstract model of computation was introduced, and we showed that it is sufficiently strong to solve any of the problems in the class NC (which includes the NP-complete problems.) We then described a number of algorithms within the model for graph-theoretic NP-complete problems. A description of the proposed implementation of our model in DNA was then presented before we concluded with a discussion of some of the factors limiting scalable DNA computation. Although this paper is theoretical in nature, we expect to test our model in the laboratory in the near future. Future work is expected to focus on an *in vitro* P-RAM simulation with $error-free$ operations.

References

[1] Leonard Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266:1021–1024, 1994.

[2] T.A. Brown. *Genetics: A molecular approach.* Chapman and Hall, 1993.

[3] James D. Watson et al. *Recombinant DNA.* Scientific American Books, 1992.

[4] Richard P. Feynman. *Miniaturization*, pages 282–296. Reinhold, 1961.

[5] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, 1988.

[6] A. M. Gibbons. *Algorithmic Graph Theory.* Cambridge University Press, 1985.

[7] David K. Gifford. On the path to computation with DNA. *Science*, 266:993–994, 1994.

[8] Juris Hartmanis. On the weight of computations. *Bulletin of the European Association For Theoretical Computer Science*, 55:136–138, 1995.

[9] Peter Kaplan, Guillermo Cecchi, and Albert Libchaber. Molecular computation: Adleman's experiment repeated. Technical report, NEC Research Institute, 1995.

[10] Richard J. Lipton. DNA solution of hard computational problems. *Science*, 268:542–545, 1995.

[11] Kary B. Mullis. The unusual origin of the polymerase chain reaction. *Scientific American*, 262:36–43, 1990.

[12] Kary B. Mullis, François Ferré, and Richard A. Gibbs, editors. *The polymerase chain reaction.* Birkhauser, 1994.

[13] Robert Pool. A boom in plans for DNA computing. *Science*, 268:498–499, 1995.

[14] John H. Reif. Parallel molecular computation: Models and simulations. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Santa Barbara, June 1995.

[15] Paul W. K. Rothemund. A DNA and restriction enzyme implementation of Turing Machines. Unpublished manuscript, 1995.

[16] Warren D. Smith and Allan Schweitzer. DNA computers in vitro and vivo. Technical report, NEC, 1995. Manuscript of 3/20/95, presented at DIMACS Workshop on DNA Based Computing, Princeton, 4/4/95.

[17] J. Williams, A. Ceccarelli, and N. Spurr. *Genetic Engineering.* βios Scientific Publishers, 1993.